by
Ray Duncan

Power Programming

DOS Extenders Old and New: Protected-Mode Programming in DOS

The agonizingly slow acceptance of OS/2, the invention of DOS extenders, the release of *Windows* 3.0, and the industry standardization of the DOS Protected Mode Interface (DPMI) all have given MS-DOS a new lease on life. While no one denies that desktop PCs will eventually migrate to a true protected-mode operating system, it's no longer so clear just what that system will be or how soon it will arrive. Although I have personally invested a lot of time and effort in OS/2 and admire its design and implementation, I've become very skeptical about its chances of replacing DOS on a large scale. It now seems much more likely that MS-DOS will simply be succeeded by itself in the form of a high-tech, 32-bit, 386-specific MS-DOS kernel running *Windows*.

In any event, DOS extender technology is the name of the game for the foreseeable future. I predict that as competition in the DOS market grows increasingly fierce, the programs become more and more complex, and users' expectations grow greater and greater, nearly every software developer will find himself or herself reaching for a DOS extender to improve performance and relieve memory limitations. I also predict that the use of expanded memory (LIM EMS) in new applications will rapidly fade away; most implementations of DPMI on 386 or later CPUs will support true virtual memory, so that EMS and the convoluted programming it requires will soon be perceived as irrelevant and not worth the effort.

Until now, DOS extenders have been relatively expensive programming environments with primitive debugging support, and they haven't been readily available to the casual programmer. This will soon change drastically, because *Windows* 3.0 has a built-in DOS extender, and all *Windows* programs that run in protected mode use it automatically. As I demonstrated in the last column, non-*Windows* applications can also take advantage of the *Windows* 3.0 DOS extender, once they've called the DPMI interface directly to switch the CPU into protected mode. In other words, many of the benefits of protected-mode programming are now potentially available to anyone who has access to a DOS-based compiler and linker and can afford $50 for the *Windows* 3.0 update.

- **It now seems likely that MS-DOS will simply be succeeded by itself in the form of a high-tech, 32-bit, 386-specific MS-DOS kernel running *Windows*, and DOS extender technology will be the name of the game for the forseeable future.**

I think it's a good idea to review how traditional DOS extenders work before taking a closer look at the *Windows* 3.0 DOS extender.

HOW TRADITIONAL DOS EXTENDERS WORK

When you buy a pre-*Windows* 3.0 DOS-based application that runs in protected mode, such as *Mathematica*, *Paradox 386*, or *Interleaf Publisher*, you're actually buying two programs: the DOS extender, which is usually licensed from a third party such as Phar Lap or Rational Systems, and the application itself. The application developer purchases a distribution license from the DOS extender developer so he can ship one executable file, with the application and DOS extender "bound" together (see Figure 1).

From the MS-DOS loader's point of view, the DOS extender is the only thing in the executable file, because it's the only part of the file that's described by the .EXE file header. MS-DOS loads the DOS extender into conventional memory in real mode, performs any necessary relocations, and passes control to the DOS extender in the normal manner. The DOS extender then allocates additional memory (conventional, extended, or both), opens its own executable file, and reads the actual application into memory, performing any fixes that might be required. MS-DOS cannot load the protected-mode application directly, because the MS-DOS loader knows absolutely nothing about protected-mode selectors and even less about the 32-bit fixes that are found in 80386 object code.

Next, the DOS extender sets up a protected-mode environment for use by the application. It builds the interrupt descriptor table, local and global descriptor tables, page tables, and task-state segment that will provide addressability once the CPU is in protected mode. This step usually includes the creation of special selectors that map the video refresh buffer, the application's program segment prefix, the BIOS data area, and other structures that the application program may wish to manipulate directly. Finally, the DOS extender switches the CPU into protected mode and passes control to the application's entry point (Figure 2).

Once the application is running, the DOS extender carries out its vital role as an interface between the protected-mode

application and real-mode DOS via its complete control of the interrupt system. In fact, the DOS extender can be thought of as a terminate-and-stay-resident (TSR) interrupt handler that—from MS-DOS's point of view—happens to live in memory owned by the application, and therefore vanishes along with the application when the application terminates. The DOS exten-der must field every interrupt that occurs in protected mode, whether it is caused by an external event (such as the real-time clock), an internal exception (such as division by zero), or a software interrupt executed by the application.

In most cases, when an interrupt occurs in protected mode, the DOS extender simply switches the CPU into real mode and "reflects" the interrupt to the original real-mode handler, but some interrupts receive special handling. For example, the DOS extender usually reprograms the 8259 interrupt controller to reassign external hardware interrupts 08h through 0Fh to new interrupt numbers, so that these interrupts can be differentiated from the internal interrupts caused by General Protection Faults, Stack Faults, and so on. The application can also register its own protected-mode handlers for internal or external interrupts, if it wishes, by calling special DOS extender functions. The most important special processing, however, occurs on the interrupts used for function requests by the application: MS-DOS INT 21h, ROM BIOS video driver INT 10h, and so on.

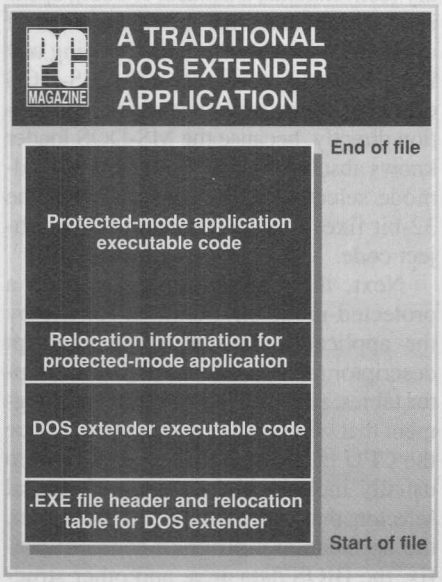

Figure 1: An example of a traditional DOS extender application. The DOS extender and the protected-mode application are bound together in a single executable file. The DOS extender is the only part of the file that is visible to the MS-DOS loader, because it is the only part described by the .EXE file header.

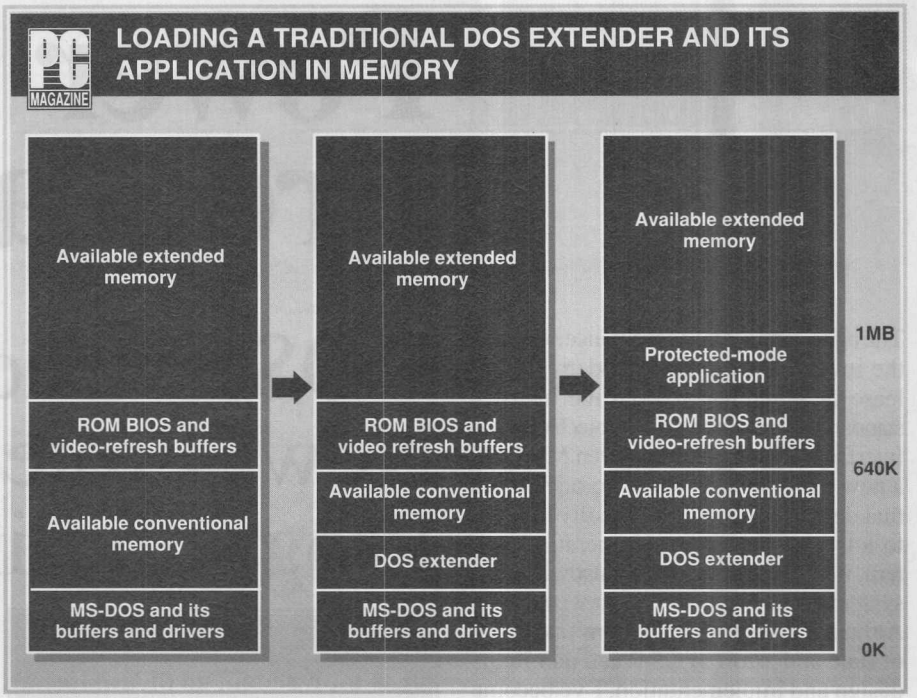

Figure 2: The MS-DOS system loader loads and relocates the DOS Extender in conventional memory. The DOS extender then allocates extended memory to hold the application, opens its own executable file, reads the application from disk, and performs any necessary fixes or relocations.

The DOS extender's actions, once it has intercepted such a function call, depend on the type of function being requested. There are four basic classes of functions that the DOS extender must be concerned with (Figure 3):

- Functions that require little more than a mode switch
- Functions that address application buffers and therefore require data movement and address translation
- Functions that must be completely replaced to make them meaningful in protected mode
- Function calls that are unique to the DOS extender itself and provide special services that have no equivalents in MS-DOS or the ROM BIOS.

In the first class of functions, all parameters are passed in registers, and the parameters don't include any addresses. The MS-DOS character I/O functions for the console, serial port, and printer are good examples. Handling of these function calls is straightforward. After it has intercepted the software interrupt, the DOS extender saves the protected-mode context, switches the CPU into real mode, and reissues the interrupt—passing control to MS-DOS or the ROM BIOS. When the function call is completed, the DOS extender switches the CPU back into protected mode, restores the full protected-mode context (such as the high 16-bits of the 80386's registers, which are not necessarily preserved by MS-DOS or its device drivers), and returns control to the application.

The next, slightly more complex class is composed of functions whose parameters include the address of a buffer or other data structure. In their original form, these addresses are meaningless to MS-DOS or the ROM BIOS for two reasons: the address is in the form of a selector and offset rather than a segment and offset, and the buffer or data almost always lies above the 1MB boundary. The DOS extender handles this problem by using its own private buffers below the 640K boundary as intermediary storage. For example, in the case of a file write request, the DOS extender intercepts the write request, copies the data from the application's buffers in extended memory to another buffer in

conventional memory, substitutes the conventional-memory buffer address for the original buffer address, switches the CPU into real mode, and finally issues the write function call to MS-DOS.

The third class of functions, where the DOS extender must replace MS-DOS or ROM BIOS services with new ones appropriate to protected mode, related mainly to memory management. For instance, the MS-DOS INT 21h Functions 48H (allocate memory block), 49h (release memory block), and 4Ah (resize memory block) are superseded by DOS extender functions that allocate, release, or resize blocks of extended memory. The DOS extender functions deal in protected-mode selectors rather than the paragraph or segment addresses used by MS-DOS, but this substitution is pretty much invisible to a "well-behaved" application, except for the much larger amounts of memory that become available in protected mode.

The final class of functions—services that are made available uniquely by the DOS extender—cover a broad spectrum. For example, there are function calls to translate between real-mode and protected-mode addresses, alter various fields of descriptors, manipulate the interrupt descriptor table, allow a direct call on a real-mode subroutine from a protected-mode application, allocate chunks of conventional memory, and so on. Most of these function calls are for highly specialized or demanding situations and are not used in typical application programs.

As you can imagine, the authors of the first DOS extenders had to be eclectic programming virtuosos of the highest order, because they were really writing little protected-mode operating systems that had to survive in an eminently hostile environment. Not only did these developers have to know the MS-DOS and ROM BIOS interfaces (documented or otherwise) inside and out, and be familiar with all the nasty little tricks of ill-behaved but indispensable applications such as *Lotus 1-2-3* or *SideKick,* they had to be complete masters of the various PC-compatible hardware architectures, of the special chip features of the 80286 and 80386, and even of the peculiarities of the various 80286 and 80386 accelerator boards.

As DPMI servers other than *Windows* 3.0 become widely available, DOS extenders will be relatively easy to implement, because all of the hardware-dependent chores such as interrupt management, mode-switching, and descriptor and page-table maintenance become the responsibility of the DPMI server. I expect that, over the next few years, we'll see DPMI-based DOS extender functionality incorporated into the runtime libraries of all the popular high-level languages, so that any program you compile will automatically run in protected mode and use extended memory if a DPMI server is present. But in the interim, the task of the DOS extender author is more arcane than ever. Not only must a commercially viable DOS extender be able to run on a bare-bones DOS system, it must also be able to coexist peacefully in a system with an XMS driver, a VCPI server, or a DPMI server (Figure 4).

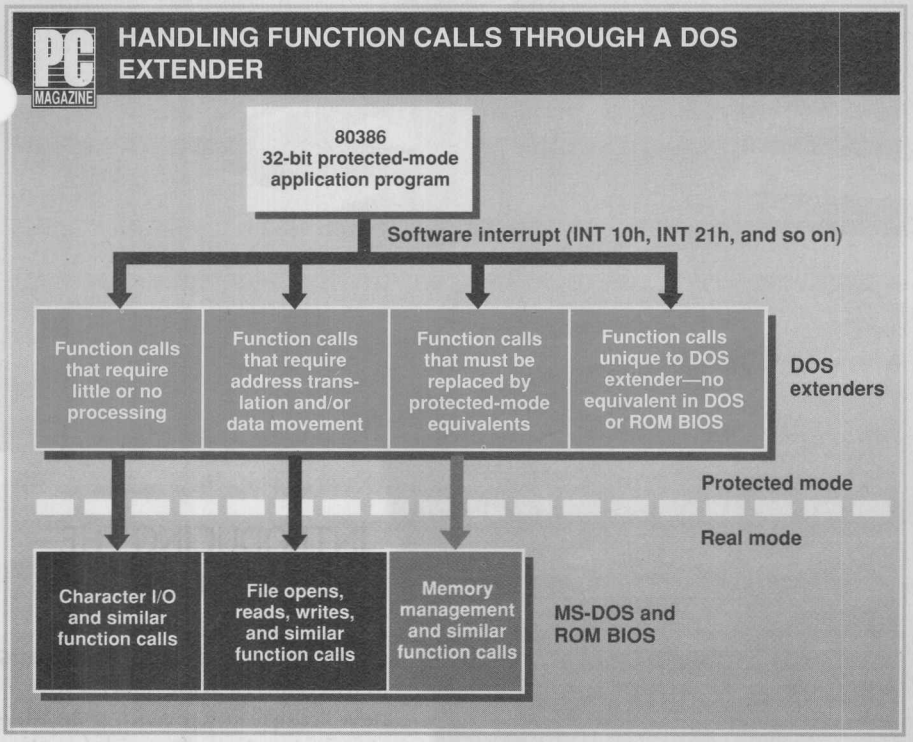

Figure 3: The function calls made by a protected-mode application fall into four different classes for handling by the DOS extender. Register-based functions can be passed straight through to DOS. Functions that have the address of a buffer or other data structure as one of the parameters require address translation and copying of the data through an intermediary buffer below the 1MB boundary. The MS-DOS memory management functions are replaced by DOS Extender functions that allocate, resize, and release blocks of extended memory. There is also a fourth group of functions that are unique to the particular DOS extender being used and have no equivalent in MS-DOS or the ROM BIOS.

THE WINDOWS 3.0 DOS EXTENDER

In *Windows* 3.0, a DOS extender is an integral part of the environment, rather than being embedded in individual application executable files like a traditional DOS extender. By using this approach—in effect, presenting programs with the same function call interface whether they are running in real mode or protected mode—Microsoft avoided many of the problems that have plagued OS/2 since its introduction. Programmers aren't confronted with a new API to learn, no new documentation has to be created, and runtime libraries don't need to be rewritten. The transition for software developers who have "followed the rules" in coding their *Windows* applications is essentially painless.

Windows 3.0's DOS extender is often confused with the fact that *Windows* 3.0 is also a DPMI server—in fact, it is the only commercially available DPMI server as this is written. The confusion has a perfectly logical explanation. Microsoft's original internal specification defined the DPMI in two layers: a set of low-level functions for interrupt management, mode switching, and extended memory man-

agement, and a higher-level interface (the DOS extender) that provided access to MS-DOS, ROM BIOS, and mouse-driver functionality via protected-mode execution of INT 21h, INT 10h, INT 33h, and so on. The higher-level functions were implemented in terms of the low-level functions and the underlying real-mode DOS and ROM BIOS services.

If the DPMI specification had become public in anything like its original form, the inevitable result would have been a de facto protected-mode version of MS-DOS. This didn't come to pass, however, because one of the trade-offs Microsoft had to make to get the existing DOS extender vendors to support DPMI was to excise its DOS extender–like capabilities. In other words, *Windows* 3.0 still has its DOS extender, but those facilities are no longer part of the DPMI specification, so the programmer cannot assume they will be present on DPMI servers other than *Windows* 3.0. The DPMI as we know it today is a sort of second-generation version of the previous industry standard, the Virtual Control Program Interface (VCPI), and its functions are not intended to serve directly as a platform for an application.

At present, the *Windows* 3.0 DOS extender is in a never-never land of semisupport and semidocumentation—like MS-DOS's TSR functions and network redirector interface. This is a bitter pill to swallow. I assume that the DOS extender documentation in the original Microsoft DPMI specification is still valid, since *Windows* 3.0 was very close to its final form at the time that document was printed, but unfortunately that document was supplied under a nondisclosure agreement and the information in it hasn't appeared publicly elsewhere. The only public documentation of *Windows* 3.0's DOS extender that I've stumbled on is a five-page leaflet that was given out at a recent *Windows* ISV seminar. This terse essay, entitled "Windows INT 21h and NetBIOS Support for DPMI," leaves a lot of interesting questions unanswered, but it does provide us with some critical information.

The first thing we learn from the document is that, although *Windows* 3.0 in 386 enhanced mode supports the full range of DPMI 0.9 functions, in standard mode *Windows* 3.0 supports only the subset of DPMI 0.9 functions that allow protected-mode programs to communicate with real-mode TSRs and device drivers:

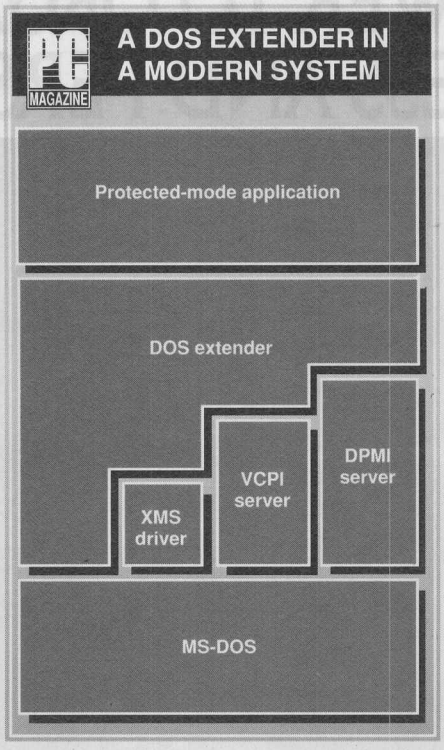

Figure 4: A modern DOS extender must be able to perform all protected-mode hardware control functions on its own, but must also be able to coexist peacefully with an XMS driver (such as HIMEM.SYS), a VCPI server (such as *QEMM-386* or *386MAX*), or a DPMI server (such as *Windows* 3.0). Since all three of these latter environments have drastically different programming interfaces and characteristics, the life of a DOS extender author is not an easy one!

DPMI Function Number	DPMI Function Name
0200h	Get Real Mode Interrupt Vector
0201h	Set Real Mode Interrupt Vector
0300h	Simulate Real Mode Interrupt
0302h	Call Real Mode Procedure with Far Return Frame
0303h	Call Real Mode Procedure with IRET Frame
0304h	Allocate Real Mode Call-Back Address
0305h	Free Real Mode Call-Back Address

The second important assertion in this document is that the *Windows* 3.0 DOS extender supports nearly the entire "official" MS-DOS application program interface in a transparent manner. The only calls listed as not being supported are

Int 20h	Get Real Mode Interrupt Vector
Int 25h	Absolute Disk Read
Int 26h	Absolute Disk Write
Int 27h	Terminate and Stay Resident

and the following obsolete INT 21h functions:

Int 21h Function	MS-DOS Function Name
00h	Terminate Process
0Fh	Open File with FCB
10h	Close File with FCB
14h	Sequential Read with FCB
15h	Sequential Write with FCB
16h	Create File with FCB
21h	Random Read with FCB
22h	Random Write with FCB
23h	Get File Size with FCB
24h	Set Relative FCB Record
27h	Block Read with FCB
28h	Block Write with FCB

A few MS-DOS functions are listed as "partially supported," but the restrictions aren't very important:

- INT 21h, functions 25h and 35h (Get and Set Interrupt Vector) only change or return the address of the protected-mode owner of the interrupt; they don't affect the handler that receives interrupts in real mode.
- Callers of INT 21h, functions 38h and 65h (Get Country Information) must be aware that the DWORD address for the case-mapping routine returned in the data structure is a real-mode address.
- INT 21h, function 44h (IOCTL), subfunctions 2, 3, 4, and 5 restrict data sizes to less than 4K if the buffer is above the 1MB boundary, while the code page capabilities of IOCTL subfunction 0Ch are not supported at all.

What about ROM BIOS calls? The original Microsoft DPMI specification explicitly listed nearly every ROM BIOS call and whether it was supported or not supported. The five-page public docum says only the following: "If a softw interrupt API is completely register-based without any pointers, segment registers,

or stack parameters, that API should work under *Windows* in protected mode.'' Fortunately, nearly all of the commonly used Microsoft Mouse driver (INT 33h) and ROM BIOS video (INT 10h), communications (INT 14h), keyboard (INT 16h), printer (INT 17h), and time and date (INT 1Ah) functions fall under the register-based umbrella and can be expected to work properly.

If you work in a high-level language, protected-mode programming under *Windows* 3.0 is easy, once you've written and debugged a small routine to make the switch to protected mode by a call to the DPMI interface. If you're lucky, your compiler's runtime library won't indulge in any of the following practices that cause problems in protected mode:

Protected-mode programming under *Windows* 3.0 is easy, once you've written a routine that will switch you into that mode.

- use of segment registers for scratch storage
- performing arithmetic on the contents of segment registers
- direct comparison of far pointers
- reading or writing outside the limits of an allocated memory object
- self-modifying code.

In the next installment, we'll continue our discussion of protected-mode programming in the *Windows* 3.0 and DPMI environments.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:

PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■